Branches and Boundaries

17 травня 2024 р.

[1]:
```
import subprocess
from itertools import product
from math import prod as mul
from random import randrange

import numpy as np
import pandas as pd
from multiprocess import Pool
```

1 Комп’ютерний практикум №1

Виконано студентами

групи ФІ-32мн

Карловський Володимир

Кріпака Ілля

1.1 Мета лабораторної роботи

Практично ознайомитися із сучасним методом криптоаналізу блокових шифрів, набути навички у дослідженні стійкості блокових шифрів до диференціального криптоаналізу.

1.2 Постановка задачі

Постановка задачі	Зроблено
Реалізувати функції шифру Хейса	+
Пошук високоімовірних п’ятираундових диференціалів шифру Хейса	+
Реалізувати атаку на сьомий раундовий ключ Хейса	+

1.3 Хід роботи / Опис труднощів

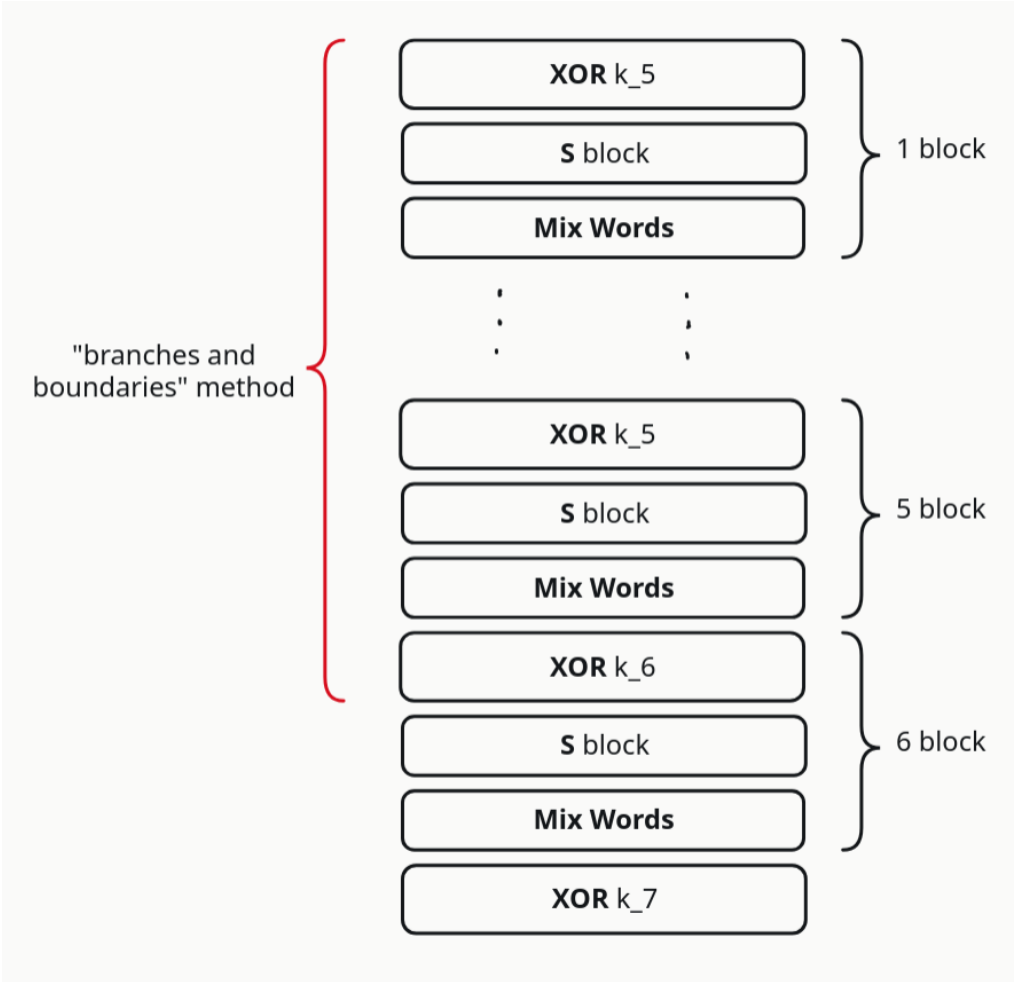
На початку практикуму разом думали яку мову краще вибрати для виконання лабораторної роботи і зійшлися на Python, через те, що на ній просто буде легше реалізувати практикум. В основному проблеми виникали із:

1. нестрогою типізацією Python, іноді не було зрозуміло де який тип і як їх можна створювати;

2. розпаралелюванням коду для швидшого виконання. Виникла проблема із простим розумінням як це можна зробити у Python, але, після прочитання інструкцій, стало зрозуміло;
3. розумінням того як треба рахувати значення ймовірностей для атаки, так як у схемі деякі кроки були пропущенні;
4. розумінням самих позначень, на схемі, а саме букв у деяких випадках;
5. розумінням того на який ключ і скільки ітерацій треба для використання методу “гілок та границь”.

Усі незрозумілості розвіялися із тим моментом, як почасли робити пошук ключів.

Диференціальний аналіз, як ми знаємо, відноситься до так званих *атак останнього раунду*, оскільки основною метою проведення аналізу є встановлення раундового ключа останнього раунду k_r. Даний метод для атаки марковського шифру реалізує евристичний пошук диференціалів за допомогою часткової побудови множини вкладених диференціальних характеристик. Основна ідея цього алгоритму полягає в тому, що ми для заданої вхідної різниці послідовно шукаємо можливі вихідні різниці на кожному раунді, але ті різниці, імовірність яких є малою (тобто нижче встановленого порогового значення), ми відкидаємо. Саму царину пошуку диференціальних характеристик можна окреслити ось такою картинкою.



Червоною дужкою позначено область роботи алгоритму для пошуку диференціальних ймовірностей. Ось, наведімо ще одну картинку для пояснення кроків алгоритму.

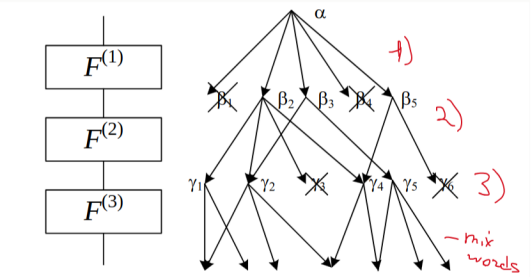
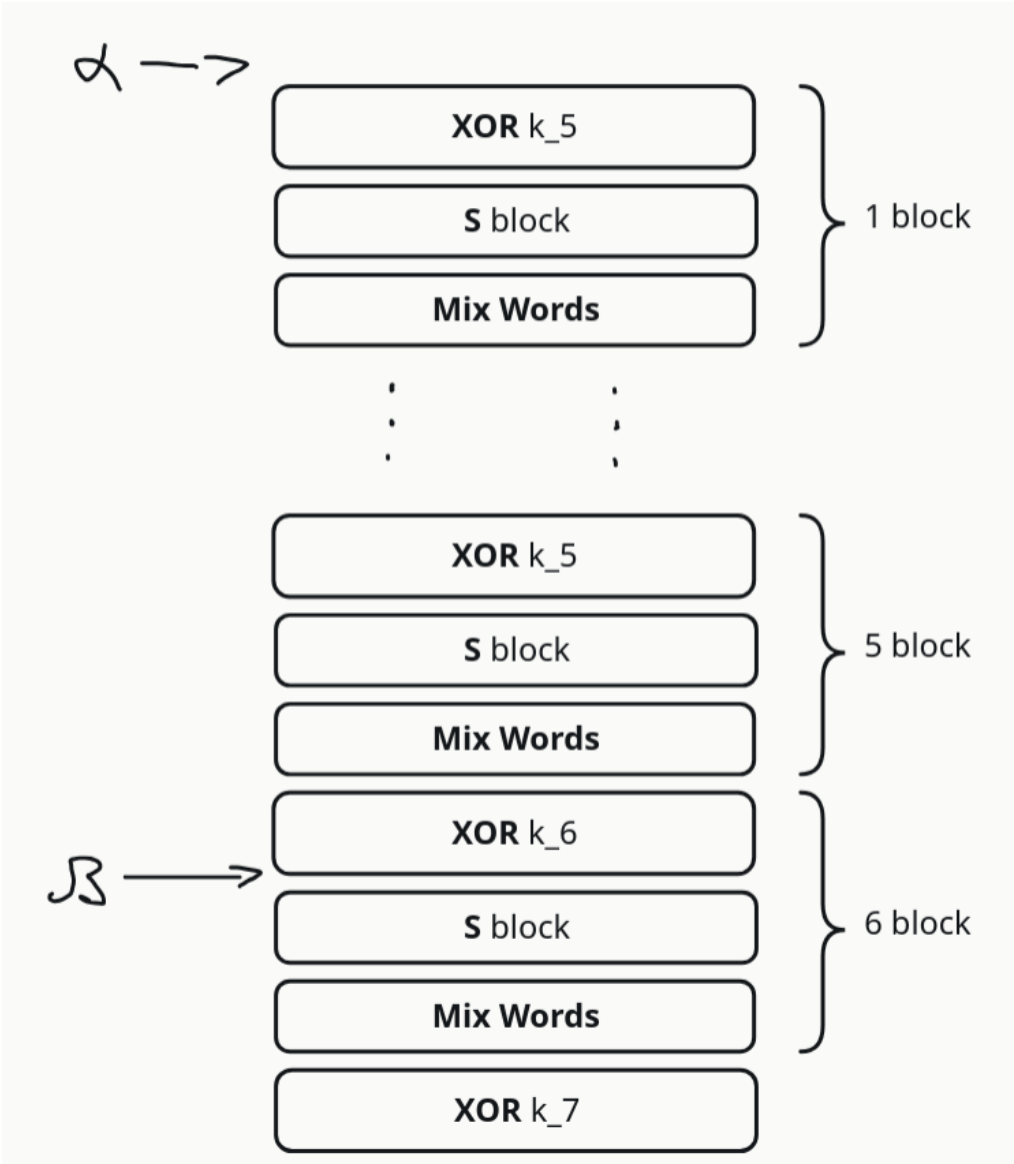


Рисунок 2.1 – Ілюстраційне представлення методу «гілок та границь»

Хід алгоритму:
1. $\Gamma_0(\alpha) := \{(\alpha, 1)\}$.
2. Для всіх t від 1 до r виконати:
3. $\quad \Gamma_t(\alpha) := \varnothing$.
4. $\quad$ Для кожної пари $(\beta_i, p_i) \in \Gamma_{t-1}(\alpha)$ виконати: // побудова «гілок»
5. $\quad\quad$ Для кожної пари $(\gamma_j, q_j) \in \Delta(\beta_i)$ виконати:
6. $\quad\quad\quad$ Якщо $(\gamma_j, p(\gamma_j)) \in \Gamma_t(\alpha)$, то:
7. $\quad\quad\quad\quad p(\gamma_j) = p(\gamma_j) + p_i \cdot q_j$;
8. $\quad\quad\quad$ інакше:
9. $\quad\quad\quad\quad$ включити у $\Gamma_t(\alpha)$ пару $(\gamma_j, p_i \cdot q_j)$.
10. $\quad$ Для кожної пари $(\gamma_i, p_i) \in \Gamma_t(\alpha)$ виконати: // перевірка «границь»
11. $\quad\quad$ Якщо $p_i \le p^*$, то:
12. $\quad\quad\quad$ вилучити пару (γ_i, p_i) з $\Gamma_t(\alpha)$.

У результаті виконання алгоритму ми отримуємо пари такого вигляду, які будуть зручні для аналізу: $((\alpha, \beta), prob)$. На наступній картинці наочно проілюстровано звідки беруться α та β.



1.4 Пояснення алгоритму

Сам алгоритм можна розбити на такі кроки:

1. ***Передобчислити таблицю диференціальних ймовірностей для кожного значення*** α, β. Саме ця таблиця дасть нам розуміння того які початкові значення α будуть переходити у відповідні значення β. І тут виникне одразу запитання чому саме ця таблиця буде використовуватися для усієї атаки? Відповідно до властивості марковості шифру Хейса та самої блокової структури шифру можна, наперед, обчислити можливі значення у які інші значення будуть переходити.
2. Переходячи до самого алгоритму ***"гілок та границь"*** його можна розділити на такі кроки:
   (а) **"Побудова гілок"**. На цьому кроці треба побудувати гілки, а саме із першої таблиці витягнути значення по яким α може перейти у відповідні β.
   (б) **Обчислення ймовірностей**. На цьому кроці треба обчислити ймовірності для диференціалів. У загальному $Pr(\alpha \to \beta) = p_i * q_i$, де p_i – ймовірність отримання певної α, а q_i – ймовірність диференціалу обчисленого із чотирьох частинок. За умови наявності диференціалу у списку, наведена ймовірність додається до попередньо обчисленого значення.
   (в) **"Відсіювання"**. На цьому кроці треба перевірити усі ймовірності диференціалів, що були обраховані на цій ітерації, зберегти їх та для вихідних значень γ застовувати heys.mix_words.
3. ***Знаходження останнього 7 раундового ключа***:
   (а) **Накопичення статистичного матеріалу**. Накопичуємо пари відкритих текстів $(X, X^{'})$, де $X^{'} = X \oplus \alpha$. Шифруємо їх та отримуємо тексти $(C, C^{'})$ відповідно.
   (б) **Обчислення кількості успішних випробувань**. Для кожного гіпотетичного ключа k_r треба розшифрувати пари $(C, C^{'})$ та отримати пари значень $(Y, Y^{'})$. (Під розшифровуванням мається на увазі [$\oplus k\_r, heys.mixWords, heys.revSubstitution$]) Далі перевіряємо гіпотезу $Y \oplus Y^{'} = \beta$. Зауважимо, що кандидати та їх (α, β) заздалегідь відомо.
   (в) **Фільтрування кандидатів**. Вибираємо кандидатів із максимальною кількістю успішних випробувань.

1.5 Порогові значення

Порогові значення для цієї лабораторної роботи були вибрані наступні.

- Поріг ймовірностей був виставлений рівним "0.0005". Так як ймовірні були в основному представлені так:
  - Максимальне значення 0.001;
  - Дуже багато значень після 0.0005 >= 0.005 (0.0002, 0.0001, …).
- поріг для накопичення статистики для знаходження 7 раундового ключа рівний 7000. Чому саме таке значення? Випадково, оскільки дозволяє залізо обчислити за притомний час усі можливі варіанти, тому обрали такі значення.

1.6 Варіант 5

S = (F, 8, E, 9, 7, 2, 0, D, C, 6, 1, 5, B, 4, 3, A)

[16]:
```
s_block = [0xF, 0x8, 0xE, 0x9, 0x7, 0x2, 0x0, 0xD, 0xC, 0x6, 0x1, 0x5,
  ↪0xB, 0x4, 0x3, 0xA]
s_block_inverse = [0x6, 0xA, 0x5, 0xE, 0xD, 0xB, 0x9, 0x4, 0x1, 0x3,
  ↪0xF, 0xC, 0x8, 0x7, 0x2, 0x0]
ENC = True
DEC = False
MAX_16BIT_NUM = (1 << 16) - 1
VARIANT = 5


def get_hex(l):
    return '[{}]'.format(', '.join(hex(x) for x in l))


def get_hex_tuple_0(l):
    return '[{}]'.format(', '.join((hex(x[0])) for x in l))


class heys:

    def heys_round(self, n, key=0):
        ct = n ^ key
        ct = self.substitute(ct, s_block)
        ct = self.mix_words(ct)
        return ct

    # r = (s_1(y_1), s_2(y_2), ... , s_n(y_n))
    def substitute(self, n, enc):
        s = s_block if enc == True else s_block_inverse
        r = 0
        for i in range(4):
            r |= s[(n >> (i * 4)) & 15] << (i * 4)
        return r

    # і-тий біт j-того фрагменту стає j-тим бітом і-того фрагменту.
    def mix_words(self, n):
        r = 0
        for j in range(4):
            for i in range(4):
                r |= (n >> (4 * j + i) & 1) << (4 * i + j)
        return r
```

1.7 Створення таблиці диференціалів

[17]:
```
heys_obj = heys()
# таблиця диференціалів для перестановки шифру Хейса
diff_substitution = np.zeros((16, 16))

for alpha in range(16):
    for beta in range(16):
        for x in range(16):
            diff_substitution[alpha][beta] += heys_obj.substitute(x ^
 ↪alpha, ENC) == heys_obj.substitute(x, ENC) ^ beta

diff_substitution /= 16

pd.DataFrame(diff_substitution)
```

[17]:

	0	1	2	3	4	5	6	7	8	9	10
0	1.0	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
1	0.0	0.000	0.000	0.000	0.125	0.125	0.000	0.250	0.000	0.125	0.125
2	0.0	0.250	0.000	0.125	0.000	0.000	0.000	0.125	0.125	0.000	0.000
3	0.0	0.125	0.125	0.000	0.000	0.000	0.250	0.250	0.000	0.125	0.125
4	0.0	0.000	0.250	0.000	0.125	0.000	0.000	0.125	0.125	0.000	0.125
5	0.0	0.000	0.000	0.125	0.000	0.000	0.125	0.000	0.125	0.125	0.000
6	0.0	0.125	0.000	0.000	0.000	0.125	0.000	0.000	0.000	0.125	0.125
7	0.0	0.000	0.125	0.000	0.000	0.250	0.125	0.000	0.125	0.000	0.000
8	0.0	0.000	0.000	0.250	0.000	0.000	0.125	0.125	0.000	0.000	0.000
9	0.0	0.000	0.000	0.125	0.125	0.000	0.000	0.000	0.125	0.250	0.125
10	0.0	0.000	0.125	0.000	0.125	0.000	0.000	0.000	0.125	0.125	0.000
11	0.0	0.125	0.000	0.000	0.125	0.125	0.125	0.000	0.125	0.125	0.125
12	0.0	0.125	0.000	0.125	0.250	0.000	0.000	0.000	0.125	0.000	0.000
13	0.0	0.125	0.000	0.125	0.125	0.125	0.000	0.000	0.000	0.000	0.125

```
14  0.0  0.000  0.125  0.000  0.000  0.125  0.125  0.125  0.000  0.000
  ↪ 0.000
15  0.0  0.125  0.250  0.125  0.000  0.125  0.125  0.000  0.000  0.000
  ↪ 0.125
```

	11	12	13	14	15
0	0.000	0.000	0.000	0.000	0.000
1	0.000	0.000	0.125	0.000	0.125
2	0.000	0.000	0.125	0.125	0.125
3	0.000	0.000	0.000	0.000	0.000
4	0.000	0.000	0.000	0.125	0.125
5	0.125	0.000	0.250	0.000	0.125
6	0.125	0.125	0.000	0.000	0.250
7	0.000	0.125	0.000	0.250	0.000
8	0.000	0.250	0.000	0.125	0.125
9	0.125	0.000	0.000	0.125	0.000
10	0.125	0.000	0.125	0.125	0.125
11	0.000	0.000	0.125	0.000	0.000
12	0.125	0.125	0.125	0.000	0.000
13	0.125	0.125	0.000	0.125	0.000
14	0.125	0.250	0.125	0.000	0.000
15	0.125	0.000	0.000	0.000	0.000

[18]:
```python
PROBABILITY_THRESHOLD = 0.0005
local_diff_table = {}


# betas
def get_new_betas(alpha):
    # obtain raw alpha values from one big Alpha
    alpha_list = [(alpha >> (4 * i)) & 15 for i in range(4)]
    # obtain non-null betas from s block diff table
    non_null_betas = []
    for i, alpha in enumerate(alpha_list):
        # filter betas with non-zero entries
        betas = [beta for beta in range(16) if
  ↪diff_substitution[alpha][beta] != 0]
        non_null_betas.append(betas)

    betas = {}
    # generate all possible combinations that can appear in resulted
  ↪beta
    for (i, beta_list) in enumerate(product(*non_null_betas,
  ↪repeat=1)):
        beta = sum([beta_list[i] << (i * 4) for i in range(4)])
```

```
        betas[beta] =
  ↪mul([diff_substitution[alpha_list[i]][beta_list[i]] for i in
  ↪range(4)])

    return betas


def differential_search(alpha):
    # shared between workers variable
    global local_diff_table

    # Γ_{t-1}(alpha) (Γ spells like 'g')
    g_prev = {alpha: 1}
    # like in rust -- 1..=5
    for i in range(1, 6):
        g_current = {}
        # misuse of 'beta' naming (by now we are iterating over last
  ↪alphas right before calculation of probabilities)
        for beta, p_i in g_prev.items():

            if beta not in local_diff_table:
                local_diff_table[beta] = get_new_betas(beta)

            # extract possible candidates for specific beta ('alpha')
  ↪and iterate over them
            for gamma in local_diff_table[beta]:
                tmp_prob = p_i * local_diff_table[beta][gamma]
                if gamma not in g_current:
                    g_current[gamma] = tmp_prob
                else:
                    g_current[gamma] += tmp_prob

        # Γ_{t}(alpha)
        g_new_filtered = {}
        #check ”bounds” and write to new list mixed words (aka gammas
  ↪are becoming new alphas)
        for gamma in g_current.keys():
            if g_current[gamma] > PROBABILITY_THRESHOLD:
                g_new_filtered[heys_obj.mix_words(gamma)] =
  ↪g_current[gamma]

        g_prev = g_new_filtered

    return (alpha, g_prev)
```

1.8 Знайдені за допомогою методу «гілок та границь» високоймовірні диференціали

Вони наведені у формі $(\alpha \to [(\beta, prob) \dots], \dots)$

[19]:
```
%%time

def prepare_alphas():
    alfas = []
    for i in range(4):
        for j in range(1, 16):
            alfas.append(j << 4 * i)
    return alfas


def init_worker(shared_diff_table):
    global local_diff_table
    local_diff_table = shared_diff_table


with Pool(initializer=init_worker, initargs=(local_diff_table,)) as
  ↪pool:
    candidates = list(
        pool.map(
            differential_search,
            prepare_alphas()
        )
    )

print('Possible candidates after differential search: ')
list(candidates)
```

```
Possible candidates after differential search:
CPU times: user 278 ms, sys: 73.4 ms, total: 351 ms
Wall time: 3.67 s
```

[19]:
```
[(1, {}),
 (2, {}),
 (3, {}),
 (4,
  {546: 0.0011348724365234375,
   8738: 0.0007147789001464844,
   1092: 0.0005817413330078125,
   2184: 0.00070953369140625,
   34: 0.0005588531494140625}),
 (5, {}),
 (6, {}),
 (7, {}),
 (8, {}),
```

```
(9, {}),
(10, {546: 0.0006341934204101562}),
(11, {}),
(12,
 {546: 0.0006477832794189453,
  8192: 0.0005099773406982422,
  8736: 0.0006477832794189453,
  8738: 0.0006477832794189453,
  2: 0.0005965232849121094,
  34: 0.0005738735198974609}),
(13, {}),
(14, {}),
(15,
 {546: 0.0008308887481689453,
  8738: 0.0005227327346801758,
  2184: 0.0005819797515869141}),
(16, {}),
(32, {}),
(48, {}),
(64,
 {1092: 0.000911712646484375,
  17476: 0.000667572021484375,
  68: 0.00054931640625}),
(80, {}),
(96, {}),
(112, {}),
(128, {}),
(144, {}),
(160, {1092: 0.0006103515625}),
(176, {}),
(192,
 {1092: 0.000621795654296875,
  17472: 0.000621795654296875,
  17476: 0.000621795654296875,
  4: 0.000518798828125,
  68: 0.00054168701171875}),
(208, {}),
(224, {1092: 0.0005340576171875}),
(240, {1092: 0.0008120536804199219, 17476: 0.0005242824554443359}),
(256,
 {1: 0.000579833984375,
  17: 0.0007476806640625,
  273: 0.00070953369140625,
  4368: 0.00070953369140625,
  4369: 0.00070953369140625,
  136: 0.0005614757537841797,
  2184: 0.0005376338958740234,
```

```
  34944: 0.0005376338958740234,
  34952: 0.0005376338958740234}),
(512,
 {16: 0.000637054443359375,
  256: 0.0005283355712890625,
  4096: 0.0005283355712890625}),
(768,
 {256: 0.00055694580078125,
  273: 0.0009918212890625,
  4112: 0.00055694580078125,
  4369: 0.0006866455078125,
  2184: 0.0007662773132324219,
  34952: 0.000522613525390625}),
(1024,
 {256: 0.000774383544921875,
  273: 0.00206756591796875,
  4097: 0.0006103515625,
  4112: 0.000762939453125,
  4353: 0.0008449554443359375,
  4369: 0.00133514404296875,
  2048: 0.0005078315734863281,
  2184: 0.0014445781707763672,
  32896: 0.0005078315734863281,
  34824: 0.0006246566772460938,
  34952: 0.001047372817993164,
  1: 0.0007686614990234375,
  272: 0.00064849853515625,
  17: 0.0011653900146484375,
  4352: 0.0008754730224609375,
  4368: 0.00084686279296875,
  136: 0.0008516311645507812,
  34816: 0.0006241798400878906,
  34944: 0.0006501674652099609,
  546: 0.0011004209518432617,
  8738: 0.0006964206695556641,
  34: 0.0005578994750976562,
  4096: 0.0005092620849609375}),
(1280, {}),
(1536, {16: 0.0005035400390625}),
(1792,
 {273: 0.0010528564453125,
  4369: 0.000518798828125,
  2184: 0.0006465911865234375,
  17: 0.00051116943359375,
  4352: 0.00051116943359375}),
(2048, {273: 0.00057220458984375}),
(2304,
```

```
 {1: 0.000766754150390625,
  17: 0.000820159912109375,
  273: 0.00090789794921875,
  4096: 0.000598907470703125,
  4368: 0.00075531005859375,
  4369: 0.00075531005859375,
  136: 0.0005474090576171875,
  2184: 0.0005562305450439453,
  34944: 0.0005562305450439453,
  34952: 0.0005562305450439453,
  16: 0.000518798828125,
  4352: 0.000507354736328125}),
(2560,
 {256: 0.000560760498046875,
  273: 0.001422882080078125,
  4112: 0.0005283355712890625,
  4353: 0.0005779266357421875,
  4369: 0.0009613037109375,
  2184: 0.0009505748748779297,
  34952: 0.0007436275482177734,
  1: 0.00067138671875,
  17: 0.0008831024169921875,
  4352: 0.0006313323974609375,
  4368: 0.000701904296875,
  136: 0.0006337165832519531,
  34944: 0.0005366802215576172,
  546: 0.0006225109100341797}),
(2816,
 {16: 0.000782012939453125,
  256: 0.0006008148193359375,
  273: 0.000957489013671875,
  4096: 0.000804901123046875,
  4368: 0.0009479522705078125,
  4369: 0.0008792877197265625,
  2184: 0.0006151199340820312,
  34944: 0.0006151199340820312,
  34952: 0.0006151199340820312,
  17: 0.0008411407470703125,
  1: 0.000698089599609375,
  136: 0.0005359649658203125}),
(3072,
 {16: 0.0010585784912109375,
  256: 0.000762939453125,
  273: 0.001537322998046875,
  4096: 0.0011348724365234375,
  4112: 0.0005893707275390625,
  4368: 0.001346588134765625,
```

```
  4369: 0.001422882080078125,
  128: 0.0006766319274902344,
  2184: 0.0009708404541015625,
  32768: 0.0007545948028564453,
  34944: 0.0009708404541015625,
  34952: 0.0009708404541015625,
  4353: 0.0007686614990234375,
  546: 0.0006389617919921875,
  8736: 0.0006389617919921875,
  8738: 0.0006389617919921875,
  1: 0.001178741455078125,
  17: 0.0011539459228515625,
  8: 0.0008325576782226562,
  136: 0.0008487701416015625,
  2: 0.0005655288696289062,
  34: 0.0005626678466796875,
  4352: 0.0006351470947265625}),
(3328,
 {16: 0.0007572174072265625,
  273: 0.00101470947265625,
  4096: 0.0007266998291015625,
  4368: 0.0009479522705078125,
  4369: 0.0010166168212890625,
  128: 0.0005192756652832031,
  2184: 0.0006649494171142578,
  32768: 0.0005018711090087891,
  34944: 0.0006649494171142578,
  34952: 0.0006649494171142578,
  1: 0.0006256103515625,
  17: 0.00075531005859375,
  136: 0.0005683898925781125}),
(3584,
 {273: 0.001068115234375,
  4369: 0.0007171630859375,
  2184: 0.0008022785186767578,
  34952: 0.0005342960357666016}),
(3840,
 {256: 0.0007801055908203125,
  273: 0.0017242431640625,
  4112: 0.0008487701416015625,
  4368: 0.0005359649658203125,
  4369: 0.00102996826171875,
  2048: 0.0005695819854736328,
  2184: 0.0012636184692382812,
  32896: 0.0005695819854736328,
  34952: 0.0008084774017333984,
  17: 0.0006885528564453125,
```

```
  4097: 0.000751495361328125,
  546: 0.0008525848388671875,
  8738: 0.0005362033843994141,
  257: 0.00055694580078125,
  4353: 0.00055694580078125,
  4352: 0.0006256103515625}),
 (4096, {}),
 (8192, {}),
 (12288, {}),
 (16384, {}),
 (20480, {}),
 (24576, {}),
 (28672, {}),
 (32768, {}),
 (36864, {}),
 (40960, {}),
 (45056, {}),
 (49152, {}),
 (53248, {}),
 (57344, {}),
 (61440, {})]
```

1.9 Відфільтровані кандидати

Вони наведені у формі $((\alpha, \beta), prob)$

[20]:
```
filtered_candidates = []
for alpha, betas in list(candidates):
    for beta, prob in betas.items():
        filtered_candidates.append(((alpha, beta), prob))
filtered_candidates = sorted(filtered_candidates, key=lambda x: x[1],
  ↪reverse=True)
filtered_candidates
```

[20]:
```
[((1024, 273), 0.00206756591796875),
 ((3840, 273), 0.0017242431640625),
 ((3072, 273), 0.001537322998046875),
 ((1024, 2184), 0.0014445781707763672),
 ((2560, 273), 0.001422882080078125),
 ((3072, 4369), 0.001422882080078125),
 ((3072, 4368), 0.001346588134765625),
 ((1024, 4369), 0.00133514404296875),
 ((3840, 2184), 0.0012636184692382812),
 ((3072, 1), 0.001178741455078125),
 ((1024, 17), 0.0011653900146484375),
 ((3072, 17), 0.0011539459228515625),
 ((4, 546), 0.0011348724365234375),
```

```
((3072, 4096), 0.0011348724365234375),
((1024, 546), 0.0011004209518432617),
((3584, 273), 0.001068115234375),
((3072, 16), 0.0010585784912109375),
((1792, 273), 0.0010528564453125),
((1024, 34952), 0.001047372817993164),
((3840, 4369), 0.00102996826171875),
((3328, 4369), 0.0010166168212890625),
((3328, 273), 0.00101470947265625),
((768, 273), 0.0009918212890625),
((3072, 2184), 0.0009708404541015625),
((3072, 34944), 0.0009708404541015625),
((3072, 34952), 0.0009708404541015625),
((2560, 4369), 0.0009613037109375),
((2816, 273), 0.000957489013671875),
((2560, 2184), 0.0009505748748779297),
((2816, 4368), 0.0009479522705078125),
((3328, 4368), 0.0009479522705078125),
((64, 1092), 0.000911712646484375),
((2304, 273), 0.00090789794921875),
((2560, 17), 0.0008831024169921875),
((2816, 4369), 0.0008792877197265625),
((1024, 4352), 0.0008754730224609375),
((3840, 546), 0.0008525848388671875),
((1024, 136), 0.0008516311645507812),
((3072, 136), 0.0008487701416015625),
((3840, 4112), 0.0008487701416015625),
((1024, 4368), 0.00084686279296875),
((1024, 4353), 0.0008449554443359375),
((2816, 17), 0.0008411407470703125),
((3072, 8), 0.0008325576782226562),
((15, 546), 0.0008308887481689453),
((2304, 17), 0.00082015991210937 5),
((240, 1092), 0.0008120536804199219),
((3840, 34952), 0.0008084774017333984),
((2816, 4096), 0.00080490112304687 5),
((3584, 2184), 0.0008022785186767578),
((2816, 16), 0.000782012939453125),
((3840, 256), 0.0007801055908203125),
((1024, 256), 0.00077438354492187 5),
((1024, 1), 0.0007686614990234375),
((3072, 4353), 0.0007686614990234375),
((2304, 1), 0.00076675415039062 5),
((768, 2184), 0.0007662773132324219),
((1024, 4112), 0.000762939453125),
((3072, 256), 0.000762939453125),
((3328, 16), 0.0007572174072265625),
```

```
((2304, 4368), 0.00075531005859375),
((2304, 4369), 0.00075531005859375),
((3328, 17), 0.00075531005859375),
((3072, 32768), 0.0007545948028564453),
((3840, 4097), 0.00075514953613281 25),
((256, 17), 0.0007476806640625),
((2560, 34952), 0.0007436275482177734),
((3328, 4096), 0.00072669982910156 25),
((3584, 4369), 0.0007171630859375),
((4, 8738), 0.0007147789001464844),
((4, 2184), 0.00070953369140625),
((256, 273), 0.00070953369140625),
((256, 4368), 0.00070953369140625),
((256, 4369), 0.00070953369140625),
((2560, 4368), 0.000701904296875),
((2816, 1), 0.00069808959960937 5),
((1024, 8738), 0.0006964206695556641),
((3840, 17), 0.0006885528564453125),
((768, 4369), 0.0006866455078125),
((3072, 128), 0.0006766319274902344),
((2560, 1), 0.00067138671875),
((64, 17476), 0.00066675720214843 75),
((3328, 2184), 0.0006649494171142578),
((3328, 34944), 0.0006649494171142578),
((3328, 34952), 0.0006649494171142578),
((1024, 34944), 0.0006501674652099609),
((1024, 272), 0.00064849853515625),
((12, 546), 0.0006477832794189453),
((12, 8736), 0.0006477832794189453),
((12, 8738), 0.0006477832794189453),
((1792, 2184), 0.0006465911865234375),
((3072, 546), 0.0006389617919921875),
((3072, 8736), 0.0006389617919921875),
((3072, 8738), 0.0006389617919921875),
((512, 16), 0.00063705444335937 5),
((3072, 4352), 0.0006351470947265625),
((10, 546), 0.0006341934204101562),
((2560, 136), 0.0006337165832519531),
((2560, 4352), 0.0006313323974609375),
((3328, 1), 0.0006256103515625),
((3840, 4352), 0.0006256103515625),
((1024, 34824), 0.0006246566772460938),
((1024, 34816), 0.0006241798400878906),
((2560, 546), 0.0006225109100341797),
((192, 1092), 0.00062179565429687 5),
((192, 17472), 0.00062179565429687 5),
((192, 17476), 0.00062179565429687 5),
```

```
((2816, 2184), 0.0006151199340820312),
((2816, 34944), 0.0006151199340820312),
((2816, 34952), 0.0006151199340820312),
((160, 1092), 0.0006103515625),
((1024, 4097), 0.0006103515625),
((2816, 256), 0.0006008148193359375),
((2304, 4096), 0.000598907470703125),
((12, 2), 0.0005965232849121094),
((3072, 4112), 0.0005893707275390625),
((15, 2184), 0.0005819797515869141),
((4, 1092), 0.0005817413330078125),
((256, 1), 0.000579833984375),
((2560, 4353), 0.0005779266357421875),
((12, 34), 0.0005738735198974609),
((2048, 273), 0.00057220458984375),
((3840, 2048), 0.0005695819854736328),
((3840, 32896), 0.0005695819854736328),
((3328, 136), 0.000568389892578125),
((3072, 2), 0.0005655288696289062),
((3072, 34), 0.0005626678466796875),
((256, 136), 0.0005614757537841797),
((2560, 256), 0.00056076049804687 5),
((4, 34), 0.0005588531494140625),
((1024, 34), 0.0005578994750976562),
((768, 256), 0.00055694580078125),
((768, 4112), 0.00055694580078125),
((3840, 257), 0.00055694580078125),
((3840, 4353), 0.00055694580078125),
((2304, 2184), 0.0005562305450439453),
((2304, 34944), 0.0005562305450439453),
((2304, 34952), 0.0005562305450439453),
((64, 68), 0.00054931640625),
((2304, 136), 0.0005474090576171875),
((192, 68), 0.00054168701171875),
((256, 2184), 0.0005376338958740234),
((256, 34944), 0.0005376338958740234),
((256, 34952), 0.0005376338958740234),
((2560, 34944), 0.0005366802215576172),
((3840, 8738), 0.0005362033843994141),
((2816, 136), 0.0005359649658203125),
((3840, 4368), 0.0005359649658203125),
((3584, 34952), 0.0005342960357666016),
((224, 1092), 0.0005340576171875),
((512, 256), 0.0005283355712890625),
((512, 4096), 0.0005283355712890625),
((2560, 4112), 0.0005283355712890625),
((240, 17476), 0.0005242824554443359),
```

```
((15, 8738), 0.0005227327346801758),
((768, 34952), 0.000522613525390625),
((3328, 128), 0.0005192756652832031),
((192, 4), 0.000518798828125),
((1792, 4369), 0.000518798828125),
((2304, 16), 0.000518798828125),
((1792, 17), 0.00051116943359375),
((1792, 4352), 0.00051116943359375),
((12, 8192), 0.0005099773406982422),
((1024, 4096), 0.0005092620849609375),
((1024, 2048), 0.0005078315734863281),
((1024, 32896), 0.0005078315734863281),
((2304, 4352), 0.000507354736328125),
((1536, 16), 0.0005035400390625),
((3328, 32768), 0.0005018711090087891)]
```

1.10 Перевірка ключів

[21]:
```
%%time

SAMPLES_NUM = 7000


# text -- 16bit numbers
def write_file(file, text: list[int]):
    with open(file, 'wb') as f:
        for x in text:
            f.write(x.to_bytes(2, byteorder='little'))
        f.close()


def read_file(file):
    text = []
    with open(file, 'rb') as f:
        data = f.read()
        for pair in zip(data[::2], data[1::2]):
            el1, el2 = pair
            text.append((el2 << 8) | el1)
        f.close()
    return text


def get_plaintext_filename(variant: int, index: int, is_mutated:
  ↪bool):
    return './key_check/pt_{0}_{1}{2}.bin'.format(variant, index,
  ↪'_mut' if is_mutated else '')
```

```
def get_ciphertext_filename(variant: int, index: int, is_mutated:
  ↪bool):
    return './key_check/ct_{0}_{1}{2}.bin'.format(variant, index,
  ↪'_mut' if is_mutated else '')


def generate_ciphertext(alpha, samples_num, index):
    plaintext = [randrange(0, MAX_16BIT_NUM) for _ in range(0,
  ↪samples_num)]
    mutated_plaintext = [text ^ alpha for text in plaintext]

    filename_pt_nonmut = get_plaintext_filename(VARIANT, index, False)
    filename_pt_mut = get_plaintext_filename(VARIANT, index, True)
    filename_ct_nonmut = get_ciphertext_filename(VARIANT, index, False)
    filename_ct_mut = get_ciphertext_filename(VARIANT, index, True)
    write_file(filename_pt_nonmut, plaintext)
    write_file(filename_pt_mut, mutated_plaintext)

    subprocess.Popen('./data/heys.bin e {0} {1} {2}'.format(VARIANT,
  ↪filename_pt_nonmut, filename_ct_nonmut),
                     shell=True,
                     stdout=subprocess.DEVNULL, stderr=None).
  ↪communicate(timeout=10)
    subprocess.Popen('./data/heys.bin e {0} {1} {2}'.format(VARIANT,
  ↪filename_pt_mut, filename_ct_mut), shell=True,
                     stdout=subprocess.DEVNULL,
                     stderr=None).communicate(timeout=10)

    ciphertext = read_file(filename_ct_nonmut)
    ciphertext_mut = read_file(filename_ct_mut)
    return ciphertext, ciphertext_mut


def check_candidate(ct_nonmut, ct_mut, k, beta):
    success_rate = 0
    for (ct1, ct2) in zip(ct_nonmut, ct_mut):
        d1 = heys_obj.substitute(heys_obj.mix_words(ct1 ^ k), DEC)
        d2 = heys_obj.substitute(heys_obj.mix_words(ct2 ^ k), DEC)
        if d1 ^ d2 == beta:
            success_rate += 1
    return success_rate


# checking keys from 0..=((1<<16)-1)
keys = [(i, 0) for i in range(0, MAX_16BIT_NUM + 1)]
```

```
for i, ((alpha, beta), _) in enumerate(filtered_candidates):
    ct_nonmut, ct_mut = generate_ciphertext(alpha, SAMPLES_NUM, i)

    with Pool() as pool:
        success_rates = list(pool.map(lambda k:
  ↪check_candidate(ct_nonmut, ct_mut, k[0], beta), keys))

    max_rate = max(success_rates)
    key_list_with_max_rate = []
    for j, (k, _) in enumerate(keys):
        if success_rates[j] == max_rate:
            key_list_with_max_rate.append((k, success_rates[j]))
    keys = key_list_with_max_rate

    print("\n\n Iteration: {0}, keys_hex: {1}, keys: {2}, max key rate:
  ↪ {3}\n\n".format(i,
                                                                    
  ↪              get_hex_tuple_0(
                                                                    
  ↪                  keys),
                                                                    
  ↪              keys,
                                                                    
  ↪              max_rate))
    # print("\n\n Iteration: {0}, keys_hex: {1}, keys: {2}, max key
  ↪rate: {3}, success_rates: {4} \n\n".
  ↪format(i,get_hex_tuple_0(keys),keys,max_rate,success_rates))

    # if there is only one key left, no sense in iterating further
    if len(keys) == 1:
        break
```

```
 Iteration: 0, keys_hex: [0x63, 0x6b, 0xe3, 0xeb, 0x863, 0x86b, 0x8e3,
  ↪0x8eb,
0x8063, 0x806b, 0x80e3, 0x80eb, 0x8863, 0x886b, 0x88e3, 0x88eb], keys:
  ↪[(99,
28), (107, 28), (227, 28), (235, 28), (2147, 28), (2155, 28), (2275,
  ↪28), (2283,
28), (32867, 28), (32875, 28), (32995, 28), (33003, 28), (34915, 28),
  ↪(34923,
28), (35043, 28), (35051, 28)], max key rate: 28
```

```
 Iteration: 1, keys_hex: [0x63, 0x6b, 0xe3, 0xeb, 0x863, 0x86b, 0x8e3,
  ↪0x8eb,
0x8063, 0x806b, 0x80e3, 0x80eb, 0x8863, 0x886b, 0x88e3, 0x88eb], keys:
  ↪[(99,
17), (107, 17), (227, 17), (235, 17), (2147, 17), (2155, 17), (2275,
  ↪17), (2283,
17), (32867, 17), (32875, 17), (32995, 17), (33003, 17), (34915, 17),
  ↪(34923,
17), (35043, 17), (35051, 17)], max key rate: 17



 Iteration: 2, keys_hex: [0x63, 0x6b, 0xe3, 0xeb, 0x863, 0x86b, 0x8e3,
  ↪0x8eb,
0x8063, 0x806b, 0x80e3, 0x80eb, 0x8863, 0x886b, 0x88e3, 0x88eb], keys:
  ↪[(99, 7),
(107, 7), (227, 7), (235, 7), (2147, 7), (2155, 7), (2275, 7), (2283, 7),
(32867, 7), (32875, 7), (32995, 7), (33003, 7), (34915, 7), (34923, 7),
  ↪(35043,
7), (35051, 7)], max key rate: 7



 Iteration: 3, keys_hex: [0x63, 0x6b, 0xe3, 0xeb, 0x863, 0x86b, 0x8e3,
  ↪0x8eb,
0x8063, 0x806b, 0x80e3, 0x80eb, 0x8863, 0x886b, 0x88e3, 0x88eb], keys:
  ↪[(99,
13), (107, 13), (227, 13), (235, 13), (2147, 13), (2155, 13), (2275,
  ↪13), (2283,
13), (32867, 13), (32875, 13), (32995, 13), (33003, 13), (34915, 13),
  ↪(34923,
13), (35043, 13), (35051, 13)], max key rate: 13



 Iteration: 4, keys_hex: [0x63, 0x6b, 0xe3, 0xeb, 0x863, 0x86b, 0x8e3,
  ↪0x8eb,
0x8063, 0x806b, 0x80e3, 0x80eb, 0x8863, 0x886b, 0x88e3, 0x88eb], keys:
  ↪[(99, 6),
(107, 6), (227, 6), (235, 6), (2147, 6), (2155, 6), (2275, 6), (2283, 6),
(32867, 6), (32875, 6), (32995, 6), (33003, 6), (34915, 6), (34923, 6),
  ↪(35043,
```

```
6), (35051, 6)], max key rate: 6
```

```
 Iteration: 5, keys_hex: [0x86b], keys: [(2155, 20)], max key rate: 20
```

```
CPU times: user 973 ms, sys: 272 ms, total: 1.25 s
Wall time: 4min 32s
```

1.11 Результат атаки:

- k_7 = 0x86b;
- кількість шт потрібна для знаходження – 5 * 7_000 = 35_000 шт ШТ.

1.12 Висновки

За допомогою реалізації практикуму із блокових шифрів по знаходженню останнього раундового ключа для шифру Хейса дізналися на практиці, як проводяться атаки даного штибу для більших криптосистем.